5.1 Variable Reassignment and Object Mutation

So far, we have largely treated objects and variables in Python as being constant over time: once an object is created or a variable is initialized, its value has not changed during the program. This property has made it easier to reason about our code: once we set the value of the variable once, we can easily look up its value at any later point in the program.[1]

[1] Indeed, this is a fact that we take for granted in mathematics: if we say "let $x = 10$" in a calculation or proof, we expect x to keep that same value from start to finish!

However, in programs it is sometimes useful to have objects and variables change value over time. We saw one example of this last week when we studied for loops, in which both the loop variable and accumulator take on multiple values over the course of running the loop. In this section, we'll introduce two related but distinct actions in a program: *variable reassignment* and *object mutation*.

Variable reassignment

Recall that a statement of the form `___ = ___` is called an *assignment statement*, which takes a variable name on the left-hand side and an expression on the right-hand side, and assigns the value of the expression to the variable.

A **variable reassignment** is a Python action that assigns a value to a variable that already refers to a value. The most common kind of variable reassignment is with an assignment statement:

```
x = 1
x = 5  # The variable x is reassigned on this line.
```

A variable reassignment *changes which object a variable refers to*. In the above example, variable `x` changes from referring to an object representing the number `1` to an object representing `5`.

The loops that we studied last week all used variable reassignment to update the *accumulator variable* inside the loop.

```
def my_sum(nums: list[int]) -> int:
    sum_so_far = 0
    for num in nums:
```

```
        sum_so_far = sum_so_far + num
    return sum_so_far
```

At each iteration, the statement `sum_so_far = sum_so_far + num` did two things:

1. Evaluate the right-hand side (`sum_so_far + num`) using the *current* value of `sum_so_far`, obtaining a new object.
2. Reassign `sum_so_far` to refer to that new object.

This is the Python mechanism that causes `sum_so_far` to refer to the total sum at the end of the loop, which of course was the whole point of the loop! Indeed, updating loop accumulators is one of the most natural uses of variable reassignment.

This loop actually illustrates another common form of variable reassignment: reassigning the *loop variable* to a different value at each for loop iteration. For example, when we call `my_sum([10, 20, 30])`, the loop variable `num` gets assigned to the value `10`, then the value `20`, and then the value `30`.

Reassignment is independent of prior uses

Consider the following Python code snippet:

```
x = 1
y = x + 2
x = 7
```

Here, the variable `x` is reassigned to `7` on line 3. But what happens to `y`? Does it now also get "reassigned" to `9` (which is `7 + 2`), or does it stay at its original value `3`?

We can express Python's behaviour here with one simple rule: **variable reassignment only changes the immediate variable being reassigned, and does not change any other variables or objects, even ones that were defined using the variable being reassigned**. And so in the above example, `y` still refers to the value `3`, even after `x` is reassigned to `7`.

This rule might seem a bit strange at first, but is actually the simplest way that Python could execute variable reassignment: it allows programmers to reason about these assignment statements in a top-down order, without worrying that future assignment statements could affect previous ones. If we're tracing through our code carefully and read `y = x + 2`, I can safely predict the value of `y` based on the current value of `x`, without worrying about how `x` might be reassigned later in the program.

That said, there is one complication with this line of reasoning that comes up with the next form of "value change", object mutation.

Object mutation

In 4.7 Nested Loops, we saw how `product` could help us calculate the Cartesian product by accumulating all possible pairs of elements in a list. Consider a function that also accumulates values in a list:

```
def squares(nums: list[int]) -> list[int]:
    """Return a list of the squares of the given numbers."""
    squares_so_far = []

    for num in nums:
        squares_so_far = squares_so_far + [num * num]
    return squares_so_far
```

Both the `squares` and `product` functions work properly, but are rather inefficient.[2] In

[2] We'll study what we mean by "inefficient" more precisely later in this course.

`squares`, each loop iteration creates a new `list` object (a copy of the current list plus one more element at the end) and reassigns `squares_so_far` to it. It would be easier (and faster) if we could somehow reuse the same object but modify it by adding elements to it; the same applies to other collection data types like `set` and `dict` as well.

In Python, **object mutation** (often shortened to just **mutation**) is an operation that changes the value of an existing object. For example, Python's `list` data type contains several methods that **mutate** the given `list` object rather than create a new one. Here's how we could improve our `squares` implementation by using `list.append`,[3] a method that adds a

[3] Check out Appendix A.2 Python Built-In Data Types Reference for a list of methods, including mutating ones, for lists, sets, dictionaries, and more.

single value to the end of a list:

```
def squares(nums: list[int]) -> list[int]:
    """Return a list of the squares of the given numbers."""
    squares_so_far = []

    for num in nums:
        list.append(squares_so_far, num * num)
    return squares_so_far
```

Now, `squares` runs by assigning `squares_so_far` to a single list object before the loop, and then mutating that list object at each loop iteration. The outward behaviour is the same, but this code is more efficient because a bunch of new list objects are not created. To use the terminology from before, `squares_so_far` is *not* reassigned; instead, the object that it refers to gets mutated.

One final note: you might notice that the loop body calls `list.append` without an assignment statement. This is because `list.append` returns `None`, a special Python value that

indicates "no value". Just as we explored previously with the `print` function, `list.append` has a *side effect* that it mutates its `list` argument, but does not return anything.

Mutable and immutable data types

We say that a Python data type is **mutable** when it supports at least one kind of mutating operation, and **immutable** if it does not. Sets, lists, and dictionaries are all mutable data types, as are the data classes we studied in the previous chapter. All of the non-collection types we've studied—`int`, `float`, `bool`, and `str`—are immutable.

Instances of an immutable data type cannot change their value during the execution of a Python program. So for example, if we have an object representing the number `3` in Python, that object's value will *always* be 3. But remember, a variable that refers to this object might be reassigned to a different object later. This is why is is important that we differentiate between variables and objects!

`list` vs. `tuple`, and what's in a `set`

All the way back in 1.3 Representing Data in Python, we introduced two Python data types that could be used to represent ordered sequences, `list` and `tuple`. We've been using them fairly interchangeably for the past few chapters, but are now ready to discuss the difference between them. *In Python, a `list` is mutable, but a `tuple` is immutable.* For example, we can modify a `list` value by adding an element with `list.append`, but there is no equivalent `tuple.append`, nor any other mutating method on tuples.

So why bother with tuples at all? Because in Python, `sets` may only contain *immutable* objects, and `dicts` may only contain *immutable keys*. So for example, we cannot have a `set` of `sets` or `set` of `lists` in Python, but we can have a `list` of `lists`, which is why studied nested lists in the last chapter.

Of course, from a theoretical standpoint a set can have elements that are other sets! So this restriction is a quirk of Python's built-in data types that we just have to live with when using this programming language.[4]

[4] In case you're curious, there is another Python data type, `frozenset`, which is an immutable version of `set`. We just won't be using it in this course.

Reasoning about code with changing values

Variable reassignment and object mutation are distinct concepts. Reassignment will change which object a variable refers to, sometimes creating a brand new object (e.g., when we used a list accumulator in `squares`). Object mutation changes the object itself, independent of what variable(s) refer t othat object.

Yet we have presented them here in the same section because they share a fundamental similarity: they both result in variables changing values over the course of a program. To illustrate this point, consider the following hypothetical function definition:

```
def my_function(...) -> ...:
    x = 10
    y = [1, 2, 3]

    ...  # Many lines of code
    ...  # Many lines of code
    ...  # Many lines of code
    ...  # Many lines of code
    ...  # Many lines of code
    ...  # Many lines of code

    return x * len(y) + ...
```

We've included for effect a large omitted "middle" section of the function body, showing only the initialization of two local variables at the start of the function and a final return statement at the end of the function.

If the omitted code does *not* contain any variable reassignment or object mutation, then we can be sure that in the return statement, `x` still refers to `10` and `y` still refers to `[1, 2, 3]`, regardless of what other computations occurred in the omitted lines! In other words, without reassignment and mutation, these assignment statements are universal across the function body: "for all points in the body of `my_function`, `x == 10` and `y == [1, 2, 3]`." Such universal statements make our code easier to reason about, as we can determine the values of these variables from just the assignment statement that creates them.

Variable reassignment and object mutation weaken this property. For example, if we reassign `x` or `y` (e.g., `x = 100`) in the middle of the function body, the return statement obtains a different value for `x` than `10`. Similarly, if we mutate `y` (e.g., `list.append(y, 100)`), the reutn statement obtains a different value for `y` than `[1, 2, 3]`. *Introducing reassignment and mutation makes our code harder to reason about, as we need to track all changes to variable values line by line.*

Because of this, you should avoid using variable reassignment and object mutation when possible, and use them in structured code patterns like we saw with the loop accumulator pattern. Over the course of this chapter, we'll study other situations where reassignment and mutation are useful, and introduce a new memory model to help us keep track of changing variable values in our code.